SawjaCard: a static analysis tool for certifying Java Card applications

Frédéric Besson, Thomas Jensen, Pierre Vittet

Inria

Abstract This paper describes the design and implementation of a static analysis tool for certifying *Java Card* applications, according to security rules defined by the smart card industry. *Java Card* is a dialect of Java designed for programming multi-application smart cards and the tool, called *SawjaCard*, has been specialised for the particular *Java Card* programming patterns. The tool is built around a static analysis engine which uses a combination of numeric and heap analysis. It includes a model of the *Java Card* libraries and the *Java Card* firewall. The tool has been evaluated on a series of industrial applets and is shown to automate a substantial part of the validation process.

1 Introduction

Security plays a prominent role in the smart card industry, due to their extensive use in banking and telecommunication. Hence, certification of smart cards has become accepted industrial practice. Traditional certifications (*e.g.*, against the Common Criteria [1]) focus primarily on the protection mechanisms of the card's hardware and operating system. More recently, attention has been drawn to the increasing number of applications that execute on the cards and the smart card industry has elaborated a set of secure coding guidelines [20,12] that apply to *basic* applications. Basic applications are granted limited privileges and the goal of the guidelines is to ensure that they do not interfere with more sensitive (*e.g.*, banking) applications. The verification of these guidelines is done by an independent authority that analyses the code and issues a certificate of conformance (or pinpoints failures of compliance). In collaboration with a company from the smart card industry we have developed the static analysis tool *SawjaCard* that can significantly simplify and automate the validation of smart card *basic* applications.

We consider applications written in the *Java Card* language – a dialect of Java dedicated to smart cards. To be validated, an application must respect a series of secure coding rules. *SawjaCard* is designed for the certification procedure proposed by AFSCM, the French industry association for Near Field Communication (NFC) technology providers, which consists of around 65 rules in total. These rules impose requirements on how an application is programmed, how it uses the resources of the card, and what kind of exceptions are acceptable.

Our main contribution is the implementation of the first static analysis tool able to automate the validation of *basic* applications according to AFSCM rules.

Our experiments show that *SawjaCard* proves automatically 87% of the properties. This work also demonstrates that precise but informal security guidelines can be mapped to formal properties that can be checked by harvesting a static analysis result. The design of the static analysis engine is a contribution of its own: we exploit the characteristics of *Java Card* but also constraints imposed by guidelines to get a precise yet efficient analyser. In terms of static analysis, the main contribution is a novel abstract domain for identifying a variant of the object-oriented *singleton object* pattern, where the nullity of a field is used to control the execution of an allocation instruction (Section 4.2).

We first present *Java Card*, highlighting the features relevant for security validation such as the *Java Card* firewall. The security requirements are described, and we explain how they can be verified on the model obtained through static analysis. We then present the main features of the analysis engine that is at the core of the tool. This includes the above-mentioned domain for detecting singleton objects and the use of trace partitioning for identifying file accesses. The tool has been evaluated against a series of industrial applications. We explain how the tool has significantly improved the certification procedure by automating a large part of the code verification.

2 Java Card

Java Card is a software development framework dedicated to multi-application smart cards. It includes a reduced version of *Java*, and has its own binary code format dedicated to devices with limited memory and processing capabilities. Like in *Java*, a *Java Card* application or library is written as a set of packages containing classes. After a standard *Java* compilation, *Java Card* converts all the classes belonging to the same package into a so-called CAP file. The CAP file is optimised to be small and meet smart card constraints, *e.g.*, fields, methods and classes are referred via *token* integers in order to reduce the size of the file.

Java Card keeps the class-based object oriented model of *Java*. Class inheritance, interfaces and the resolution mechanism for virtual and static calls is the same as in *Java*. Primitive types are restricted to the *Java* boolean, byte, short and optionally integer (might not be supported by every platform and the use of integer is also forbidden in some security guidelines). Arrays are limited to single-dimensional arrays. Strings are not available, neither as a primitive type, nor as a provided library class. Garbage collection is not required by the *Java Card* specification and it is still common not to have a garbage collector in a smart card. As memory is precious, application are expected to allocate data only during their installation phase and as parsimoniously as possible. Security guidelines emphasise this aspect (*e.g.*, allocation in a loop is forbidden).

The bytecode language is close to the *Java* bytecode language but with some noticeable differences. It is still a stack-based language with basically a reduced instruction set. However there are some differences which makes the *Java Card* bytecode more than a strict subset. For example, the operand stack contains 16

bits values and the standard operations work on such short values. Hence, each arithmetical operation is semantically different from its Java counterpart.

2.1 Modelling the *Java Card* runtime and its libraries

Our static analysis tool performs a whole program analysis. It takes a single application CAP file but also expects additional CAP files representing all the used libraries. The core *Java Card* libraries are usually not available for analysis. They are i) proprietary; ii) card dependent; iii) and (partly) implemented using native code. To get a portable and card independent whole program, we have implemented the core *Java Card* libraries and standard extensions such as *GlobalPlatform* or ETSI standard UICC [10] using pure *Java Card* code extended by specification methods that are built-ins of the analyser: `RANDOM`, `ASSUME` and `ASSERT`. As we are not interested in proving functional correctness but security properties, the model of a method is usually simple and is based on the informal specification of the libraries. The model initialises system resources of the *Java Card* runtime. For instance, it allocates singleton exceptions objects that are owned by the Java Card runtime or global system input/output buffers. The model is also responsible for simulating the different phases of the applet life cycle. The *install* phase consists in initialising the applet state. The applet is also assigned its Application IDentifier (AID) and is registered within the *Java Card* runtime. The *process* phase is an infinite event loop where the applet processes commands. Eventually, the applet enters the *uninstall* phase where it is removed from the card.

2.2 Modelling the *Java Card* firewall

The *Java Card* security architecture relies on a *firewall* which strongly limits inter-applet communication. The firewall mechanism guarantees that an inter-applet communication triggers a dynamic run-time check whose success depends on the *owner* of the object and the running *context* of the code. Every created object is assigned an owner, which is the context of the applet performing the object allocation. Each method is assigned the context of its enclosing applet. At runtime, the virtual machine ensures that an applet can only manipulate objects in its context and raises a *SecurityException* otherwise.

Communication between applets is achieved through *Shareable* interfaces. Using a specific method of the runtime, an applet A receives a shareable request from an explicitly identified applet B. The applet A can accept the request depending on the identity of B and return an object o implementing a *Shareable* interface. When applet B calls a method of object o, a context switch occurs and the method runs with the context of A.

Our Java model makes explicit the security checks performed by the Firewall using built-in API calls to obtain the owner of objects `GET_OWNER` or the running contexts `GET_CALLER_CONTEXT` (see Fig. 1). The owner/context properties are directly modelled by the abstract domains: each abstract object in the heap is assigned a owner and the abstract call stack is tagged by running contexts.

```
/* @API javacard.framework.AID: "Throws: SecurityException
 *     - if anObject object is not accessible in the caller's context" */
public final boolean equals(Object anObject){
 short caller_ctx = GET_CALLER_CONTEXT();
 if(!JCRESystem.accessible_in_caller_context(anObject, caller_ctx))
   throw JCRESystem.securityException;
 if (anObject==null || !(anObject instanceof AID)) return false;
 return [...] }
```

Figure 1: Example of API performing a Firewall check

This precise modelling of the Firewall is necessary to rule out security exceptions. Needless to say that the validation of applets strictly forbids security exceptions.

3 Validation of Java Card applications

The validation of *Java Card* applications is based on several sets of coding guidelines, edited by industrial stakeholders. The main source of guidelines comes from the AFSCM [20], a French association of companies working in the area of NFC and contact-less smart cards. The AFSCM guidelines consists of 65 coding rules that specify properties that an applet must obey in order to be validated. Rules from the Global Platform [12] initiative have also been integrated. Some rules (such as "*The interactions between different interfaces must be clearly defined.*") are not amenable to automatic verification. Others are not considered because they concern the Java source ("*A switch statement must always include a default case.*"). Eliminating duplicates, we extracted 55 verifiable rules from the guidelines mentioned above, and classified them as shown in Fig. 2.

The rules vary significantly in granularity and type of property. Some properties are purely syntactic ("*An application must not contain a* `nop` *instruction*", or "*An application shall not use an AID already registered.*") whereas others require information about the dynamic behaviour of the application ("*no null pointer exceptions*" or "*no array-out-of bounds exceptions*"). Some rules specify

Type	Number	Examples
syntactical	20	Strictly forbidden methods, no Nop, package name checking
constant values	6	Constant array size, Proactive commands and events
call graph	6	Allocation only in specific part of the code.
exceptions	10	Ensure that various exceptions will never occur
file system	4	Report read or written files, ensure deletion of created file.
termination	1	No recursive code.
other	8	Applet instance should not be placed in static field.
total	55	

Figure 2: Classification of the rules (from AFSCM [20] and Global Platform [12]).

restrictions on how library methods can be called, and with what arguments. Most of these rules cannot be verified *as is*, exactly due to the undecidability of the underlying semantic property, and approximations are called for. As mentioned above, an important feature of these rules is that certain rules simplify the verification of others. *E.g.*, knowing that the rules "*Recursive code is forbidden*" and "*Arrays must be allocated with a determined size*" are satisfied means that the analyser can strive for more precision without running into certain complexity and computability issues. In the following, we explain how the validation of the rules can be done by mining the static analysis result.

Numeric values: In *Java Card*, resources are often accessed using integer identifiers and managed by calling methods with a fixed set of flags. Many rules specify that those integers must be constant or range over a set of legitimate values. Our analyser is computing an abstraction of numeric values and therefore of method arguments. The abstraction of the method arguments is checked for compliance with the rules.

Array values: Some resources can be coded by arrays of integers. For instance, files are identified by an array `[`$\mathtt{i}_1$`;...;`$\mathtt{i}_n$`]` which represents the file name $\mathtt{i}_1$`/.../`$\mathtt{i}_n$. Menu entries (*i.e.*, strings) are coded by arrays of bytes. As with numeric values, validation rules often require those arrays to be constant. Files are an exception. File names are constructed by a sequence of calls `fh.select(`$\mathtt{d}_1$`)` ... `fh.select(`$\mathtt{d}_n$`)` where `fh` is a `FileView` object and the $\mathtt{d}_i$ are typically constants, identifying directories. Our analyser does not track sequences of events but our model of the class `FileView` is an array representing the current working directory that is updated by calls to the `select` method. Our analyser models arrays and provides for each index a numeric abstraction of the content. This abstraction is queried in order to validate rules about resources encoded as arrays.

Control-flow: The validation rules impose constraints on the control-flow graph of the application—especially during the installation phase. For instance, most memory allocations are required only to take place during the *install* phase, identified by a call to the `install` method. The analysis is constructing an abstract control-flow graph corresponding to the inlined control-flow graph of the application. Constraints over the control-flow graph can therefore be checked by exploring the abstract control-flow graph. For the particular case of memory allocation, we traverse the graph and make sure that memory allocation primitives *e.g.*, `new` statement, are only accessible from the `install` method.

Exceptional behaviour: Validation rules are strict about exception handling. Run-time exceptions such as *ArrayOutOfBounds*, *NullPointerException* and *ClassCastException* are strictly forbidden. In our bytecode intermediate representation, run-time exceptions correspond to explicit instructions and we generate verification conditions for all those instructions. For obvious reasons, security exceptions (*SecurityException*) are also forbidden. The abstraction of the heap is designed to model object ownership and can be used to ensure that the security checks performed by the *Java Card* Firewall do no raise *SecurityException*. There are other rules about exceptional behaviours which can be

interpreted as coding guidelines. The analysis is precisely modelling the flow of exceptions. In particular, it collects for each handler the caught exception and for each method call the escaping exceptions. This information is sufficient for checking all the rules regarding exceptions.

4 Overview of the static analysis engine

Our static analysis engine is designed specifically for *Java Card* and its particular programming style. Existing general purpose analysis frameworks for *Java* *e.g.*, [32,19,17] cannot be applied directly to *Java Card*. Firstly, existing frameworks do not provide a CAP front-end – this is a non-negligible engineering issue. Although CAP files are compiled from class files, the inverse transformation is far from obvious. For instance, the instruction set is different and dynamic method lookup is compiled using explicit virtual tables. Secondly, our static analysis engine exploits fully the fact that *Java Card* programs are relatively small, forbid recursion and allocate few objects. Standard *Java* analyses designed to scale for object-oriented programs cannot exploit this. Finally, the *Java Card* firewall which has no *Java* counterpart is also modelled directly at the analysis level.

Our analyser operates on a 3-address code intermediate bytecode representation A3Bir [7] that is obtained by pre-processing the binary CAP file. This representation is adapted from the *Sawja* framework [17] and has the advantage of making explicit the different runtime checks performed by the *Java Card* Virtual Machine. An example of such intermediate code is given Fig. 4.

The static analysis engine implements an inter-procedural scheme which consists in a dynamic inlining of method calls. The benefit is a precise inter-procedural scheme that mimics the behaviour of a concrete interpreter. In terms of abstract domains, the domain of the inter-procedural analysis is D^* given that D is the domain for the intra-procedural analysis. This approach is effective for two reasons that are specific to *Java Card*: recursion is forbidden and the programs are relatively small.

4.1 Combining intervals, constant sets and symbolic equalities

To abstract numeric values, the analyser is using a combination of three abstract domains: intervals, finite sets and symbolic equalities. The code snippet of Fig 3 illustrates their collaboration. The abstract domain of intervals *Int* is very popular but is not precise enough if used alone. At Line 4 of Fig. 3, the interval [0;5] is used to abstract the non-consecutive values 0 and 5. As many *Java Card* methods take as arguments integer flags or return integer flags that are in general not consecutive, this is a common source of imprecision. To deal with this issue, we use a domain $Fin = \mathcal{P}(Cst) \cup \{\top\}$ where Cst is the set of constants that appear in the program text. For efficiency, information about these constants is lost after arithmetic operations and reductions with the interval domain are limited to the cases of $\bot$ and singletons *i.e.*, abstract elements with represent a single concrete value. For Line 4, the product $Int \times Fin$ gives a precise invariant but

```
1  byte i = 0;      /*i:[0;0] & {0}*/
2  if(RANDOM_BOOL()){
3   i = 5;          /*i:[5;5] & {5}*/
4  }                /*i:[0;5] & {0;5} */
5  byte j = i;      /*i:[0;5] & {0,5} j:[0;5] & {0,5} with j == i */
6  i  = (j==0) ? (byte)(i+5) : i; /*i:[5;5] & {5} j:[0;5] & {0,5} */
```

Figure 3: Collaboration between numeric abstract domains

would still be unable to infer that at Line 6 the value of `i` can only be 5. To get this result, it is necessary to propagate through the test `j==0` the knowledge that `j` equals `i`. This is a known weakness of *non-relational* domains which compute an abstraction for each variable independently. There are well-known numeric relational domains *e.g.*, convex polyhedra [6], octagons [24]. These domains are very expressive but are also computationally costly. Our analyser is using a more cost-effective weakly relational domain [25] computing for each program point and each local variable x an equality $x = e$ where e is a side-effect free expression of our intermediate representation *i.e.,* an expression built upon variables, arithmetic operators and field accesses. At Line 5, we have the equality `j==i`. Hence, when `j==0`, `i` is also 0 and when `j != 0`, `j` have value 5 and so has `i`. Combined, the three domains are able to compute the precise invariant of Line 6.

Symbolic expressions improve the precision of numeric abstractions but also significantly contribute to ruling out spurious null pointers. This is illustrated by Fig. 4. Our goal is to verify that `bar` is called with a non-null argument. At source level, this property is obvious. However, the cumulative effect of Java compilation, CAP conversion and the re-construction of our analyser intermediate representation introduces temporary variables. Without symbolic expressions, at Line 3, we only know that the temporary variable `t0` is not null but this variable is not used anymore. Symbolic expressions keep track of equalities between variables `t0, t1, t2`, the constant `null` and the value of field `this.foo`. Using the theory of equality, we can deduce at Line 2 that `this.foo` is not null. This information is propagated at Line 4 where we deduce that `t2` is also not null. Therefore, at the call in Line 5, `t2` is still not null.

```
1  void baz(){
2   if(foo!=null)
3    bar(foo);}
```

```
1  t0=this.foo; t1=null; if (t0==t1) goto 5;
2   /* t0==this.foo & t1==null */
3   t2=this.foo; checknotnull (this!=null);
4   /* t0==this.foo & t1==null & t2==this.foo */
5  this.bar(t2);   return;
```

Figure 4: Left: Source code Right: A3Bir representation

4.2 Points-to analysis with detection of singleton objects

Our heap abstraction is *flow-sensitive* and takes the form of a *points-to* graph [15]. A node in the graph is an abstract object identified by a call-stack and the creation point of the `new` statement, or a special node representing the `null` value. Object creation is done with parsimony in *Java Card*, and most `new` statements only ever allocate *one* object. In other words, most abstract objects are singletons representing a single concrete object. If an abstract object is a *singleton*, the analyser can precisely model side effects and perform *strong updates*.

Abstract counting of the number of allocations of an abstract object $o \in AO$ is a standard technique for identifying singletons objects. The properties "not allocated", "allocated at most once" and "allocated many times" are encoded by the numeric domain $\{0, 1, \infty\}$. Our domain is therefore $Cnt = AO \to \{0, 1, \infty\}$. In *Java Card* most objects are allocated during the *install* phase. This phase is loop-free and abstract counting therefore precisely identifies singleton objects. However, abstract counting fails at identifying singleton objects that are allocated during the *process* command loop.

To improve precision we propose a novel abstract domain able to capture variants of the so-called object-oriented singleton pattern. The idea behind the singleton pattern is that a field *fd* is either i) `null` and the singleton object is not allocated or ii) not `null` and the singleton object is allocated. To capture such conditional invariants, our singleton domain maps each field to a valuation of the abstract allocation counters. The abstract domain is therefore $Sgton = Field \to Cnt$. Consider $sgton \in Sgton$ and a field *fd*. The intuition is that $sgton(fd)$ only provides information about abstract counters under the condition that the field *fd* is null. When the points-to ensures that a field *fd* is definitely not `null`, the condition does not holds and $sgton(fd)$ can be set arbitrarily, in particular it can be strengthened to $\bot$. When the points-to ensures that a field *fd* is definitely `null`, the condition holds and $sgton(fd)$ can be used to strengthen the current abstract counters. If the condition cannot be decided, the information $sgton(fd)$ remains dormant but becomes useful after *e.g.*, a test $fd = null$ or would be updated after an assignment to the field *fd*.

Formally, given a concretisation $\gamma_{Cnt} : Cnt \to \mathcal{P}(Heap)$ for the *Cnt* domain, the concretisation $\gamma : Sgton \to \mathcal{P}(Heap)$ is defined by:

$$h \in \gamma(sgton) \text{ iff } \bigwedge_{fd} h(fd) = null \Rightarrow h \in \gamma_{Cnt}(sgton(fd)).$$

Fig. 5 illustrates how the reduced product [4] of a points-to, *Sgton* and *Cnt* analyses can ensure the singleton property. Before the loop (Line 1), the object *o* is not allocated, the field *fd* is definitely `null` and the conditional property ($\texttt{fd} = \texttt{null} \Rightarrow (\texttt{o} \mapsto \texttt{0})$) holds. At the head of the loop (Line 4), the object *o* is a singleton, the field *fd* is either `null` or points to the object *o*. The singleton domain holds the key invariant: the object *o* is not allocated when the field *fd* is `null`. After the test `fd == null` (Line 6), we refine our points-to and conclude that *fd* is definitely `null`. Therefore, the conditional property of the singleton domain holds: we can exploit the right hand side of the condition and refine the

```
1    /* fd = null & (fd = null ⇒ (o ↦ 0)) & o ↦ 0 */
2    [...]
3    while(true){
4      /* fd ∈ {null, o} & (fd = null ⇒ (o ↦ 0)) & o ↦ 1 */
5      if (fd == null)
6      /* fd = null & (fd = null ⇒ (o ↦ 0)) & o ↦ 0 */
7      fd = new o();
8      /* fd = o & (fd = null ⇒ ⊥) & o ↦ 1 */      }
```

Figure 5: Singleton pattern

abstract counter ($o \mapsto 0 \sqcap o \mapsto 1 = o \mapsto 0$) and conclude that the object o is not allocated. After the allocation of o (Line 8), the object o is a singleton ($o \mapsto 1$) and *fd* points to o. Hence, *fd* is definitely not `null`. This is where the conditional singleton domain becomes useful. Because the condition $fd = null$ now no longer applies, the abstract counters can be strengthened to $\bot$. For simplicity, say that the abstract state after the (empty) `else` branch is the abstract state of the loop head of Line 4. At the end of the conditional, after the join, we get the same abstract state as at the loop head, which is therefore a fixpoint.

In practice, our *Sgton* domain also maintains conditions over instance fields and numeric fields (`false` plays the role of `null`). For our *Java Card* applications, those enhancements allow a precise identification of all singleton objects.

5 File system access: a case for trace partitioning

Trace partitioning [27] is a generic technique for locally improving the precision of a static analysis. It consists in partitioning an abstract state depending on a history of events. Suppose that the original abstract state is $D^\sharp$, after trace partitioning, the abstract state is $(Event^* \times D^\sharp)^*$ where *Event* is an arbitrary set of syntactic events (*e.g.*, a call to a specific method) or semantic events (*e.g.*, the variable x has value v). We have successfully used trace partitioning for precisely determining the files accessed by an application.

As explained in Section 3, *Java Card* comes with a hierarchical file system. In our model, the current directory $\texttt{i}_1\texttt{/.../i}_n$ is coded by an array of short $\texttt{[i}_1\texttt{;...;i}_n\texttt{]}$ that is stored in the `path` field of a file handler object `fh` implementing the `FileView` interface. Moving to the $\texttt{i}_{n+1}$ directory is done by the method call $\texttt{fh.select(i}_{n+1}\texttt{)}$. Therefore, determining the accessed files requires a precise analysis of the array content.

Consider the code of Fig. 6 that is representative of how files are accessed in *Java Card*. Suppose that before calling the `cd` method, the field `fh` is either `null` or points to an object `ofh` such that $\forall \texttt{i}, \texttt{ofh.path[i]} = 0$. At the method return, with our base abstraction, the effect of the three paths is merged. We loose precision and get $\texttt{res} = \texttt{fh} \in \{\texttt{null}; \texttt{ofh}\} \wedge \texttt{ofh.path}[0] \in [0; 1] \wedge \texttt{ofh.path}[1] \in [0; 20]$. However, the precise post-condition of the `cd` method is $\texttt{P}_1 \vee \texttt{P}_2 \vee \texttt{P}_3$ where each

```
1 public static FileView cd(){
2  if (fh!=null){
3   fh.select((short)1);
4   if(RANDOM_BOOL()){return null;}
5   fh.select((short)20); }
6  return fh; }
```

Figure 6: Typical code for accessing files

P_i models the effect of a particular execution path.

$$\mathtt{P_1} \triangleq \mathtt{res} = \mathtt{null} \wedge \mathtt{fh} = \mathtt{null} \wedge \mathtt{ofh.path}[0] = 0 \wedge \mathtt{ofh.path}[1] = 0$$
$$\mathtt{P_2} \triangleq \mathtt{res} = \mathtt{null} \wedge \mathtt{fh} = \mathtt{ofh} \wedge \mathtt{ofh.path}[0] = 1 \wedge \mathtt{ofh.path}[1] = 0$$
$$\mathtt{P_3} \triangleq \mathtt{res} = \mathtt{ofh} \wedge \mathtt{fh} = \mathtt{ofh} \wedge \mathtt{ofh.path}[0] = 1 \wedge \mathtt{ofh.path}[1] = 20$$

Even for *Java Card*, the disjunctive completion [4] of our base abstract domain does not scale. Trace partitioning [27] offers a configurable trade-off between efficiency and precision. In particular, it allows a fine-grained control of when abstract states should be merged or kept separate. Our trace partitioning strategy is attaching to abstract states the trace of the encountered `select` calls. A trace event is therefore of the form $\mathtt{select_i}$ where i identifies uniquely the method call in the control-flow graph. At the end of the `cd` method, we obtain: $[] : \mathtt{P_1} \quad [\mathtt{select_3}] : \mathtt{P_2} \quad [\mathtt{select_3}; \mathtt{select_5}] : \mathtt{P_3}$. The security guidelines mandate that the applet *install* phase and the processing of a single command of the *process* phase should terminate. We exploit this information and merge traces at those specific events. This strategy is precise and terminating for all the *Java Card* applications we have analysed.

6 Experimental evaluation

We have evaluated *SawjaCard* on 8 industrial *Java Card* applets. For confidentiality reasons, we are required to keep them anonymous. The applications are representative of basic applications. There are loyalty applets but also phone applications. They require few privileges but access nonetheless certain non-sensitive part of the file system. The characteristics of the applets can be found in Fig. 7. For each applet, we provide the number of instructions of the application and the number of instructions taking into account the libraries used by

Applet	A1	A2	A3	A4	A5	A6	A7	A8
Instrs (app)	2769	2835	1823	1399	636	752	1245	230
Instrs (+ libs)	5824	5236	4301	5643	2834	3044	3402	2040
CFG	3435	6096	1491	1247	825	999	842	487
Time	29min	19min	6min	2min	32s	18s	4s	2s

Figure 7: Applet characteristics

the application. To give a finer estimate of the complexity of the code, we also provide the number of nodes in the inlined control-flow graph constructed by *SawjaCard*. However, this is still a coarse-grained measure of code complexity which is weakly correlated with the analysis time. For instance, applet A1 executes more instructions than applet A2, has fewer CFG nodes but takes longer to analyse. The analysis time is obtained using a laptop with a Intel Core i7 processor and 8 GB of memory.

Fig. 8 summarises the analysis results for the 8 applets. We made a selection of the properties that can be evaluated in a fully automatic way *i.e.*, the result is either boolean or can be expressed as a percentage of alarms. An entry in Fig. 8 reads as follows. A ✓ denotes a fully verified property. A property is marked x if it is a true violation according to our manual inspection of the code. A ? denotes a false positive. A number denotes a percentage. For instance, 90 means that the property holds for 90% of the program points relevant for the property – percentages are rounded from below. If it is in bold red, the remaining alarms are true violation. Otherwise, we could not ascertain the absence of false positives. For 75% of the properties, *SawjaCard* reports no alarm and thus those properties are automatically validated. We have investigated the remaining alarms manually. For 12% of the properties, we have concluded that the alarms were all genuine violations of the properties. Therefore, the verdict of *SawjaCard* is precise for 87% of the properties. For the remaining 13%, there are false positives. For instance, we identified that certain *ArrayOutOfBounds* alarms were due to a lack of precision of the analysis. However, on average, *SawjaCard* validates nonetheless about 87% of array accesses. For the remaining alarms, there are false positives but also real alarms. In the following, we explain in more details a selection of the properties of Fig 8.

NullPointerException[1] The precision of our null pointer analysis is satisfactory as *SawjaCard* validates 98% of reference accesses. Moreover, for 4 of the applets, we could conclude after manual inspection that the remaining alarms were real errors. For the other 4, the reasoning is more intricate and there might still be false positives. A typical error we found consists in ignoring that certain APIs can (according to the official specification) return a null pointer in some unusual cases. For instance, the method `getEntry` of the class **uicc.toolkit.ToolkitRegistrySystem** may return null *if the server does not exist or if the server returns null*. None of the analysed application performs the necessary defensive check to protect itself against such a null pointer.

ArrayOutOfBounds[2] *SawjaCard* validates 87% of array accesses. The remaining accesses are sometimes false positives. In particular, we have identified an access whose verification would require a relational numeric analysis. However, a

[1] From AFSCM rules: A basic application shall not include any code that leads to NullPointerException, whatever this exception is caught or not.

[2] From AFSCM rules: An application must not include any code that leads to ArrayOutOfBoundException, caught or not.

Alarms	A1	A2	A3	A4	A5	A6	A7	A8
NullPointerException	94	98	**99**	99	**97**	**98**	97	**99**
ArrayOutOfBounds	71	88	92	87	92	98	90	98
CatchIndividually	**46**	**23**	**82**	**31**	**32**	**67**	**57**	**53**
CatchNonISOException	×	×	×	×	×	×	×	×
HandlerAccess	×	✓	×	×	×	✓	✓	✓
AllocSingleton	✓	✓	✓	✓	✓	×	✓	✓
SDOrGlobalRegPriv	×	✓	✓	✓	✓	✓	✓	✓
SWValid	?	✓	✓	✓	✓	✓	✓	✓
ReplyBusy	?	✓	✓	✓	✓	✓	✓	✓
ClassCastException	✓	✓	✓	✓	✓	✓	✓	✓
NegativeArraySize	✓	✓	✓	✓	✓	✓	✓	✓
ArrayStoreException	✓	✓	✓	✓	✓	✓	✓	✓
SecurityException	✓	✓	✓	✓	✓	✓	✓	✓
AppletInStaticFields	✓	✓	✓	✓	✓	✓	✓	✓
ArrayConstantSize	✓	✓	✓	✓	✓	✓	✓	✓
InitMenuEntries	✓	✓	✓	✓	✓	✓	✓	✓

Figure 8: Analysis results – selected properties.

simple rewrite of the code would also resolve the problem. Other array accesses rely on invariants that are not available to the analyser. For instance, certain array indexes are read from files. More precisely, when reading a file, certain applications first read a special segment, which is the file status. The full size of the file is a field of this file status. As the content of the file cannot be known, it is impossible to track this length.

CatchIndividually[3] This rule corresponds to a strict coding discipline that is almost syntactic: each type of exception should be caught by a different handler. This property is responsible for numerous alarms. All the reported alarms correspond to true violations of the rule. For instance, the following not compliant code snippet catches all the different exceptions with a single handler.

```
1 try{buffer = JCSystem.makeTransientByteArray((short)140,CLEAR_ON_RESET);}
2 catch (Exception e) {buffer = new byte[(short)140];}
```

CatchNonISOException[4] All the applets trigger alarms for this property. The alarms correspond to violations of the property. The exceptions that are ignored correspond to exceptions that are not thrown by the application itself but escape from library code. It might very well be that the proprietary implementations

[3] From Global Platform rules: The Application should catch each exception defined by the used APIs individually in the application code and should explicitly rethrow the exception to the card runtime environment if needed.

[4] From AFSCM rules: All exceptions thrown during the execution from any defined entry point must be caught by the application, except *ISOException* that are thrown in response to a command.

never raise these exceptions. Nonetheless, their possibility is reflected by our model of the API which is based on the *Java Card* API specification.

Other properties. The AFSCM rules forbid the classic *Java* exceptions: *ClassCastException*, *NegativeArraySize* and *ArrayStoreException*. For all the applets, *SawjaCard* proves their absence. Thanks to our modelling of *Java Card* Firewall, *SawjaCard* is also able to rule out *SecurityExceptions*. The rule *AppletInStaticFields* specifies that applet objects should not be stored in static fields. This property is validated for all the applets. The next two rules concern values that should be constant: array sizes and menu entries. Those rules are also validated for all the applets. The rule *SDOrGlobalRegPriv* is about privileges that should be granted to access certain APIs. Applet 1 requires certain privileges and therefore raises an alarm. The rules *SWValid* and *ReplyBusy* specify the range of the status word return by applets. The rule is verified for all the applets except applet 1. This is probably a false alarm given that the applet is using a non-standard way of computing the status word. The last rule concerns certain method calls returning handlers that should be protected by try-catch blocks. *SawjaCard* raises an alarm for all the applets. This rule is indeed violated.

DeadCode[5] For all the applets, *SawjaCard* detects some dead code which is due to the Java compilation. Consider the following method which unconditionally throws an exception. The *return* instruction is not reachable but is required by the *Java* compiler.

```
1 void dead_code (short val){ SystemException.throwIt(1); return; }
```

The *Java* compiler also enforces that method should list the exceptions they *might* raise using a *throws* clause. However, the algorithm for checking this clause is purely syntactic. To make *Java Card* compile, a defensive approach consists in adding handlers for all the *potential* exceptions. For certain calls, *SawjaCard* proves that certain exceptions are never thrown and that the handlers are therefore dead code. For compliance with the rule, a workaround would be to remove the useless handlers and add to the *throws* clause of the method all the exceptions that are proved impossible.

File handling [6] There is a significant number of properties concerning files. Some of them are simple and do not lead to false positives (such as *CreateFile*, *CreateFilesAtInstall*, *FileResizing* which only require to check arguments for specific method calls). Other properties are more complex and require a precise identification of the files that are read or written. Using trace partitioning (see Section 5), we can precisely identify files paths that are constructed by a sequence of select instructions with constant arguments. However, certain applets make

[5] From AFSCM rules: dead code must be deleted/removed from the code.

[6] From AFSCM rules: The file system provides access to files that are under the control of the Mobile Network Operator. These files shall not be accessed by applications, except for a few exceptions.

the assumption that the AID of a particular application is stored at a specific position in a system file. The AIDs is thereafter used to access the root of the sub file system owned by the application AID. Our model of the file system is too coarse to encode this assumption and therefore we cannot handle this pattern.

Allocation[7] The alarms are real violations. Most applets allocate objects after the *install* phase. Yet, more relaxed rules allow the allocation of singleton objects. This rule is still violated by applet 6 which repeatedly tries to get a handler. In our model of the library, each unsuccessful try allocates an intermediate object and is therefore responsible for a memory leak. For the other applets, our singleton domain is precise and ensures that memory allocation is finite.

7 Related work

For analysing Java programs, there are mature static analysis frameworks such as Soot [32] and Wala [19]. Based on Wala, the Joana tool [13] is able to prove security properties based on information-flow. Information-flow analyses would probably benefit from the *Java Card* restrictions. Currently, AFSCM guidelines do not consider such properties and are limited to safety properties.

Algorithms tuned for Java are usually not well-fitted for the constraints of *Java Card*. In particular, state-of-the-art algorithms for constructing control-flow graphs of Java programs are based on context-sensitive flow-insensitive points-to analyses [22,29]. For *Java Card*, our analyser demonstrates that a context-sensitive flow-sensitive points-to analysis is viable. It dynamically inlines methods calls and therefore literally computes an ∞-CFA. The *Astree* analyser is using a similar strategy for handling function calls [5]. In their context, the programs are large and function calls are rare. *Java Card* programs are comparatively tiny but method calls are ubiquitous.

For Java, Hubert *et al.,* [16] show how to infer the best `@NonNull` annotations for Fähnrich and Leino type system [11]. The static analyser Julia [30,31] implements a more costly but also more precise null pointer analysis that can be efficiently implemented using BDDs. Because our objects are singletons, our flow-sensitive points-to analysis performs *strong updates* and is therefore precise enough to precisely track null pointers and rule out *NullPointerExceptions*.

Might and Shivers [23] show how to improve abstract counting of objects using abstract garbage collection. Their analysis can prove that an abstract object corresponds to a single *live* concrete object. Our singleton domain is based on a different program logic and can ensure that an abstract object is only allocated once. As *Java Card* usually does not provide garbage collection, we really need to prove that there are only a finite number of allocated objects.

Semantics [28,9] and analyses [14,8] have been proposed for *Java Card*. Huisman *et al.,* [18] propose a compositional approach to ensure the absence of illicit applet interactions through *Shareable* interfaces. For *basic* applications

[7] From Global Platform rules: A basic application should not perform instantiations in places other than in install() or in the applet's constructor.

such interactions are simply forbidden. Our tool verifies that applets do not expose *Shareable* interfaces and therefore enforces a simpler but stronger isolation property. A version of the Key deductive verification framework [2] has been successfully applied to *Java Card* [26]. JACK [3] is another deductive verification tool dedicated to *Java Card* that is based on the specification language JML [21]. However, deductive verification is applied at the source level and requires annotations of the code with pre-(post-)conditions. This methodology is not applicable in our validation context which needs to be fully automatic for binary CAP files.

8 Conclusions

The validation process for smart card applications written in *Java Card* involves around 55 rules that restrict the behaviour of the applications. This process can benefit substantially from static analysis techniques, which can automate most of the required verifications, and provide machine-assistance to the certifier for the rest. The SawjaCard validation tool contains a static analysis which combines analysis techniques for numeric and heap-based computations, and which is further enhanced by specific domain constructions dedicated to the handling of the file system and *Java Card* firewall. A substantial part of building such a validation tool involves the modelling of libraries for which we propose to build a series of stubs whose behaviour approximates the corresponding APIs sufficiently well for the analysis to be accurate. Benchmarks on a series of industrial application shows that the tool can analyse such applications in a reasonable time and eliminate more than 80% of the required checks automatically.

The development of the tool suggests several avenues for further improvements. The properties for which the tool could be improved are *ArrayOutOfBoundException* and file properties. The numeric analysis is only weakly relational, and it would be possible to increase its precision by using a full-blown relational domains such as polyhedra or octagons. An effective alternative to significantly reduce the number of alarms would be to impose stricter coding rules (for example defensive checks for narrowing down the range of non constant indexes). Our model of the file system could also be improved. To get a precise and scalable analysis, our assessment is that file system specific abstract domains should be designed. Certain properties are also simply not provable because they depend on invariants that are established by the *personalisation* phase of the application. This phase happens after the *install* phase and corresponds to commands issued, in a secure environment, by the card manufacturer. Currently, the end of this phase has no standard specification and cannot be inferred from the applet code. For the others properties we have satisfactory results: when the tool emits an alarm, it corresponds to a real error in the application. The tool has been recently transferred to industry where it will be used as part of the validation process.

Acknowledgements. We thank Delphine Demange, Vincent Monfort and David Pichardie for their contributions to the development of the tool.

References

1. *Common Criteria for Information Technology Security Evaluation*, 2012.
2. W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, W. Mostowski, and P. H. Schmitt. The KeY system: Integrating Object-Oriented Design and Formal Methods. In *FASE*, volume 2306 of *LNCS*, pages 327–330. Springer, 2002.
3. G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK - A Tool for Validation of Security and Behaviour of Java Applications. In *FMCO*, volume 4709 of *LNCS*, pages 152–174. Springer, 2006.
4. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *POPL*, pages 269–282. ACM Press, 1979.
5. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ Analyzer. In *ESOP*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.
6. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–96. ACM Press, 1978.
7. D. Demange, T. P. Jensen, and D. Pichardie. A Provably Correct Stackless Intermediate Representation for Java Bytecode. In *APLAS*, volume 6461 of *LNCS*, pages 97–113. Springer, 2010.
8. M. Éluard and T. P. Jensen. Secure Object Flow Analysis for Java Card. In *CARDIS*, pages 97–110. USENIX, 2002.
9. M. Éluard, T. P. Jensen, and E. Denney. An Operational Semantics of the Java Card Firewall. In *E-smart*, volume 2140 of *LNCS*, pages 95–110. Springer, 2001.
10. ETSI Project Smart Card Platform. Smart Cards; UICC Application Programming Interface (UICC API) for Java Card™.
11. M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA*, pages 302–312. ACM, 2003.
12. GlobalPlatform Inc. *GlobalPlatform Card Composition Model Security Guidelines for Basic Applications*, 2012.
13. J. Graf, M. Hecker, and M. Mohr. Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In *ATPS'13*, volume 215 of *LNI*, pages 123–138. GI, 2013.
14. R. R. Hansen and I. Siveroni. Towards Verification of Well-Formed Transactions in Java Card Bytecode. *Electr. Notes Theor. Comput. Sci.*, 141(1):145–162, 2005.
15. M. Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *PASTE'01*, pages 54 – 61. ACM, 2001.
16. L. Hubert. A non-null annotation inferencer for Java bytecode. In *PASTE*, pages 36–42. ACM, 2008.
17. L. Hubert, N. Barré, F. Besson, D. Demange, T. P. Jensen, V. Monfort, D. Pichardie, and T. Turpin. Sawja: Static Analysis Workshop for Java. In *FoVeOOS*, volume 6528 of *LNCS*, pages 92–106. Springer, 2010.
18. M. Huisman, D. Gurov, C. Sprenger, and G. Chugunov. Checking Absence of Illicit Applet Interactions: A Case Study. In *FASE*, volume 2984 of *LNCS*, pages 84–98. Springer, 2004.
19. IBM. The T.J. Watson Libraries for Analysis (Wala). http://wala. sourceforge.net.
20. P. Le Pallec, S. Diallo, T. Simon, A. Saif, O. Briot, P. Picard, M. Bensimon, J. Devisme, and M. Eznack. *Cardlet Development Guidelines*. AFSCM, 2012.
21. G. T. Leavens, J. R. Kiniry, and E. Poll. A JML Tutorial: Modular Specification and Verification of Functional Behavior for Java. In *CAV*, volume 4590 of *LNCS*, page 37. Springer, 2007.

22. O. Lhoták and L. J. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1), 2008.
23. M. Might and O. Shivers. Improving flow analyses via GammaCFA: abstract garbage collection and counting. In *ICFP*, pages 13–25. ACM, 2006.
24. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
25. A. Miné. Symbolic Methods to Enhance the Precision of Numerical Abstract Domains. In *VMCAI*, volume 3855 of *LNCS*, pages 348–363. Springer, 2006.
26. W. Mostowski. Formalisation and Verification of Java Card Security Properties in Dynamic Logic. In *FASE*, volume 3442 of *LNCS*, pages 357–371. Springer, 2005.
27. X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.
28. I. Siveroni. Operational semantics of the Java Card Virtual Machine. *J. Log. Algebr. Program.*, 58(1-2):3–25, 2004.
29. Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL*, pages 17–30. ACM, 2011.
30. F. Spoto. The Nullness Analyser of julia. In *LPAR*, volume 6355 of *LNCS*, pages 405–424. Springer, 2010.
31. F. Spoto. Precise null-pointer analysis. *Software and System Modeling*, 10(2):219–252, 2011.
32. R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, page 13. IBM, 1999.